In unit testing, my main focus was to make sure that every test aligned with the requirements of each class. I tested important scenarios, especially boundary conditions and incorrect inputs, to ensure that every class handles these cases correctly. For instance, for the Appointment class I checked conditions like making an appointment with an ID that is too long

```
10      // Test for appointment ID that is too long
11      @Test(expected = IllegalArgumentException.class)
12      public void testAppointmentIdTooLong() {
13          new Appointment("12345678901", new Date(), "Checkup");
14      }
```

This approach ensures that the software requirements for each class are met as every test attempts to imitate possible problems that may happen in real-world scenarios.

The unit tests that I wrote, I believe are quality due to the fact that they possess a high coverage percentage. Every one of my files had a minimum coverage of 80%, implying that the tests successfully covered important pathways and extreme situations within the code. The areas with the highest coverage were those anticipated to raise exceptions, for example in for the Task class, I wrote unit tests to ensure the handling of improper task IDs and names fulfilled the requirements specified for the class.

```
54      // Tests that the description can be set correctly
55      @Test
56      public void testSetDescription() {
57          Task task = new Task("1", "Test Task", "This is a test task.");
58          task.setDescription("New Description");
59          assertEquals("New Description", task.getDescription());
60      }
```

Assertions are another validation of test effectiveness. They check the state of objects after creation and modification, making sure application's behavior matches the expected outcomes. This thorough testing method assists in discovering bugs at an early stage and improves codebase maintainability.

Writing the JUnit tests was a great learning experience, allowing me to dive deeper into technical aspects of my code. For instance, by writing tests that checked for null inputs and invalid lengths, I ensured that the code was technically sound. This code snippet in the"TaskTest”

```
10      // Test that a Task object is created correctly with valid inputs
11      @Test
12      public void testTaskCreation() {
13          Task task = new Task("1", "Test Task", "This is a test task.");
14          assertEquals("1", task.getTaskId());
15          assertEquals("Test Task", task.getName());
16          assertEquals("This is a test task.", task.getDescription());
17      }
```

tests if a task is created correctly. Additionally, this test from the same test file ensures that a user could not type a name that is too long (specified by the requirements of the Task class)

```
37      // Tests that an exception is thrown when the description is too long
38      @Test
39      public void testDescriptionLength() {
40          Exception exception = assertThrows(IllegalArgumentException.class, () -> {
41              new Task("1", "Test Task", "12345678901234567890123456789012345678901234567890123456789012345678901");
42          });
43          assertEquals("Description must not be null and must be 50 characters or less.", exception.getMessage());
44      }
```

by testing inputs where appropriate, I was able to ensure that my code was both technically sound and efficient, reducing redundancy in the tests while maximizing coverage.

In this project, I primarily used unit testing to validate individual components, such as methods or classes, ensuring that each class functioned as expected. This allowed me to focus on small, isolated parts of the application, helping to catch errors early on in the development process. For instance, I tested constraints like invalid IDs or dates to confirm that exceptions were thrown as needed. Unit testing was very effective in maintaining code quality.

Although useful, I didn’t use integration testing, which tests how different units work together. This technique could have been useful to ensure that classes like Appointment, ContactService, and Task interacted seamlessly within a larger application. While I believe unit testing was sufficient enough for this project, I now believe that integration testing could have

provided additional confidence in the interaction between all of the components. Other techniques, such as system testing and acceptance testing, focus on the overall system and user experience. These are particularly important in larger projects to ensure the software meets all requirements and is user-friendly. For this project, unit testing I believe was appropriate, but expanding to include these other techniques on top of unit testing, I believe would offer a more comprehensive evaluation of the software's overall performance and usability.

As I worked through this project, the mindset that I adopted was to be cautious and detail-oriented, as I recognized the importance of thoroughly testing each component. An event that not only made me realize the importance of testing, but also helped me adopt this mindset was the recent event that happened with CrowdStrike. As a software tester throughout this project, I approached the task with the understanding that even small errors could have significant consequences, particularly when components interact. For example, in testing the Appointment class, I did my best to carefully check edge cases, such as handling past dates and invalid IDs, to ensure that the application would behave correctly in all scenarios.

To limit bias in my review of the code that I wrote, I approached testing with the mindset of finding errors, rather than confirming correctness. This involved writing tests that challenged the assumptions made during development, such as testing for null inputs or excessively long strings. Recognizing that bias could be a concern when testing my own code, I made an effort to view the code objectively, considering how a fresh pair of eyes might identify issues I could overlook. For example, I questioned the reliability of the ContactService class by simulating real-world use cases that could lead to failure, ensuring that my tests were not only thorough but were also unbiased. Maintaining a disciplined commitment to quality is essential in software engineering, as cutting corners can lead to technical debt and long-term issues. Throughout this

project, I resisted the temptation to skip thorough testing, as I understand that quality code requires careful attention to detail. To avoid technical debt in the future, I plan to consistently apply best practices, such as regular code reviews, comprehensive testing, and iterative development. I am so very thankful for this course as it has taught me so much about testing. With the lessons that I learned throughout this course, I aim to build software that is not only functional, easier to maintain, and reliable long term.